Formal Methods for Software Development

Till Mossakowski, Lutz Schröder

25.10.2004

Why Testing?

- Usually, testing is not closely integrated with development.
- This prevents you from measuring the progress of development — you can't tell when something starts working or when something stops working.
- Programmers know they should write tests for their code. Few do: "I'm in too much of a hurry."
- Vicious cycle: more pressure $\Rightarrow$ less time for tests $\Rightarrow$ less productive/code less stable $\Rightarrow$ more pressure.
- $\Rightarrow$ burn out!

The Point of a Testing Framework

- Breaking out this vicious cycle requires an outside influence
- $\Rightarrow$ a simple testing framework makes a big difference.
- incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts.
- testing cycle: encounter new bugs $\Rightarrow$ catch them with tests (immediate feedback!) $\Rightarrow$ fix them.

Testing with user-defined test cases (HUnit)

In the Haskell module where your tests will reside, import module HUnit:

```
import HUnit
```

Define test cases as appropriate:

```
test1 =
 TestCase (assertEqual "for (foo 3)," (1,2) (foo 3))
test2 = TestCase (do
   (x,y) <- partA 3
   assertEqual "first result of partA," 5 x
   b <- partB y
   assertBool ("(partB " ++ show y ++ ") failed") b)
```

Executing the Tests

Name the test cases and group them together:

```
tests = TestList [TestLabel "test1" test1,
                  TestLabel "test2" test2]
```

Run the tests as a group:

```
> runTestTT tests
Cases: 2  Tried: 2  Errors: 0  Failures: 0
>
```

Sample Program

```
foo x = (x-2,x)

partA x = do putStrLn ("\npartA: "++show x)
             return (x+2,x-2)
partB x = do putStrLn ("\npartB: "++show x)
             return (x==1)
```

More About Monads in Haskell

Christoph Lüth:
Fortgeschrittene Konzepte der funktionalen Programmierung

Mi. 17-19h, MZH 8090

`http://www.tzi.de/~cxl/lehre/asp.ws04/`

Sorting Example

```
insert :: Ord a => (a,[a]) -> [a]
insert(x,[]) = [x]
insert(x,y:l) = if x <= y then x:y:l
                          else y:insert(x,l)

insert_sort :: Ord a => [a] -> [a]
insert_sort([]) = []
insert_sort(x:l) = insert(x,insert_sort(l))
```

Test Cases

```
testSorting
  = TestCase
      (do let list = [7,2,6,3,5]
              sortedList = [2,3,5,6,7]
          assertBool "insert_sort is faulty"
            (insert_sort list == sortedList)
      )
```

Installation of HUnit

Glasgow Haskell Compiler:
`http://www.haskell.org/ghc`

HUnit:
`http://hunit.sourceforge.net/`
bunch of Haskell files, put them in your local Haskell directory

Writing Tests

```
type Assertion = IO ()

assertFailure :: String -> Assertion
assertFailure msg =
    ioError (userError ("HUnit:" ++ msg))
```

- Assertions ensure properties and may raise exceptions.
- Assertions are combined to make a test case.
- Test cases are combined into tests.

More Assertion Functions

```
assertBool :: String -> Bool -> Assertion
assertBool msg b = unless b (assertFailure msg)

assertString :: String -> Assertion
assertString s = unless (null s) (assertFailure s)

assertEqual :: (Eq a, Show a) =>
                  String -> a -> a -> Assertion
assertEqual preface expected actual =
  unless (actual == expected) (assertFailure msg)
 where msg = (if null preface then ""
                  else preface ++ "\n") ++
              "expected: " ++ show expected ++
              "\n but got: " ++ show actual
```

Combining Assertions

- Assertions may be combined, using do notation.
- The result is a single, collective assertion, which fails if any of its constituent assertions is executed and fails.
- By contrast, distinct test cases are executed independently. The failure of one is independent of the failure of any other.

Test Cases

- `(TestCase (return ()))`
  is a test case that never fails.
- `(TestCase (assertEqual "for x," 3 x))`
  is a test case that checks that the value of `x` is 3.
- The structure of a Test is a tree:

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

- Using a hierarchy helps organize tests just as it helps organize files in a file system.

Recognizing Tests

The number of test cases that a test comprises can be computed with testCaseCount.

```
testCaseCount :: Test -> Int
```

A test is identified by its path in the test hierarchy.

```
data Node  = ListItem Int | Label String
  deriving (Eq, Show, Read)

type Path = [Node]
  -- Node order is from test case to root
testCasePaths :: Test -> [Path]
```

Advanced Features

HUnit provides additional features for specifying assertions and tests more conveniently and concisely. These facilities make use of Haskell type classes.

```
infix 1 @?, @=?, @?=
(@?) :: (AssertionPredicable t) =>
          t -> String -> Assertion
pred @? msg = assertionPredicate pred
                  >>= assertBool msg
(@=?) :: (Eq a, Show a) => a -> a -> Assertion
expected @=? actual = assertEqual "" expected actual
(@?=) :: (Eq a, Show a) => a -> a -> Assertion
actual @?= expected = assertEqual "" expected actual
```

The Type Class AssertionPredicable

```
type AssertionPredicate = IO Bool

class AssertionPredicable t
 where assertionPredicate :: t -> AssertionPredicate

instance AssertionPredicable Bool
 where assertionPredicate = return

instance (AssertionPredicable t) =>
          AssertionPredicable (IO t)
 where assertionPredicate = (>>= assertionPredicate)
```

The Type Class Assertable

```
class Assertable t
 where assert :: t -> Assertion

instance Assertable ()
 where assert = return

instance Assertable Bool
 where assert = assertBool ""

instance (ListAssertable t) => Assertable [t]
 where assert = listAssert
```

```
instance (Assertable t) => Assertable (IO t)
 where assert = (>>= assert)

class ListAssertable t
 where listAssert :: [t] -> Assertion

instance ListAssertable Char
 where listAssert = assertString
```

The Type Class Testable

```
class Testable t
 where test :: t -> Test

instance Testable Test
 where test = id

instance (Assertable t) => Testable (IO t)
 where test = TestCase . assert

instance (Testable t) => Testable [t]
 where test = TestList . map test
```

More Convenient Notation

```
infix  1 ~?, ~=?, ~?=
infixr 0 ~:
(~?) :: (AssertionPredicable t) => t -> String -> Test
pred ~? msg = TestCase (pred @? msg)

(~=?) :: (Eq a, Show a) => a -> a -> Test
expected ~=? actual = TestCase (expected @=? actual)

(~?=) :: (Eq a, Show a) => a -> a -> Test
actual ~?= expected = TestCase (actual @?= expected)

(~:) :: (Testable t) => String -> t -> Test
label ~: t = TestLabel label (test t)
```

Example for More Convenient Notation

```
tests = test [
 "test1" ~: "(foo 3)" ~: (1,2) ~=? (foo 3),
 "test2" ~: do
   (x, y) <- partA 3
   assertEqual "for the first result of partA," 5 x
   partB y @? "(partB " ++ show y ++ ") failed" ]
```

Pros and Cons for HUnit

Pros:

- quickly applicable
- no knowledge about data domains necessary
- test run automatically

Cons:

- user has to supply test cases
- degree of coverage is unclear
- only executable properties can be tested